Impact of Hardware/Software Partitioning upon the System Testing Cost

Ghassan Al-Hayek, Yves Le-Traon & Chantal Robach

LSR-IMAG, 46 Avenue Félix Viallet, 83031 Grenoble, France

Phone: (33) 04 76 57 46 88 Fax: (33) 04 76 57 47 17

E-mail: {Ghassan.Al-Hayek, Yves.Le-Traon, Chantal.Robach}@imag.fr

Abstract

This paper studies the impact of hardware/software partitioning upon the testing cost of codesign systems. The testability of a hierarchical specification is discussed and an estimate proposed to evaluate the cost of system testing. It depends on cost values of hardware/software testing for each unit-level component. These values are provided by a mutation-testing approach applied to both software and hardware implementations. Results show that this approach provides a new helpful partitioning criterion which can be used together with other already known criteria. A case study provided by AEROSPATIALE illustrates this testing cost oriented partitioning.

1. Introduction

Most often, application specific systems include a mixture of hardware and software components. During the design process, the hardware/software (HW/SW) partitioning of system functionality constitutes a delicate phase which has a decisive impact on the final cost of system testing. Partitioning consists in dividing system functionality into groups and deciding for each one whether it will be implemented in software (using off-the-shelf microprocessors) or in hardware (using custom circuits such as ASIC or FPGA components). Many recent works have proposed ways to overcome this major difficulty. They are essentially based on performance optimization. Most of them propose a compromise between better performances and lower cost designs [1-4] .

However, none of them suggest how to optimize testability, which, at this stage is roughly defined as the property of the system to be easily tested.

To assist the designer, appropriate metrics are used to drive the partitioning operation as soon as the description of the design is available. To our knowledge, no testability metric exists to guide partitioning at the specification stage of codesign systems. Paradoxically, the sooner the testability problems are tackled, the more decisive the impact on test costs. From this point of view, testability-based partitioning is helpful to the designer and has a considerable impact on the final product. In addition, research work on testability does not currently address the particular problem of combining software and hardware testability metrics. It focuses only on either hardware circuits [5, 6] or software components [7, 8] without enabling comparisons. In [9] , the principles for analyzing a codesign system architecture are detailed, but no way for integrating the specificity of hardware components is provided.

In this paper, we consider hierarchically specified systems, with unit-level components at the lowest level. Partitioning of such systems consists, for each unit-level component, in choosing either a hardware or software implementation. For evaluating the testing cost, a common scale is needed for comparing the testing cost of hardware and software parts. To overcome this difficulty, we relate testing cost to the number of cases needed to test the system functionality. The basic idea of the approach is to recursively evaluating the number of cases needed for testing each specification level. The testing cost is highly affected by two parameters:

- unit-level components implementation and test,
- test strategies of upper level specifications.

At the unit-level, the common scale is obtained using a unique test methodology efficient on both hardware and software implementations: mutation-based testing technique. Originally proposed for software testing [10], mutation testing has been adapted to efficiently test hardware implementations [11]. As a result, we obtain the number of test cases which is relevant to the testing cost of unit level components.

At upper levels, we consider various structural strategies of system testing. All-Paths, Start-Small and Start-Big strategies are applied and a correlated testability analysis technique is proposed which has been specifically developed for data flow codesign.

This approach is illustrated on design specifications of embedded software from the avionics industry. The case study is provided by AEROSPATIALE* in the form of a Computer Assisted Specification diagram, representing data flows. It corresponds to the most common description mode for designers in avionics. Test strategies and testability analysis are based on a model which, in a single graph, highlights relationships between data (variables and constants) and helps identifying the partial functions of the specification This graph, the Information Transfer Graph, models information transfers throughout the system. Using the graph, information flows are computed from a given data flow specification and test strategies are specified.

Section 2 details: the state of the art concerning the problem of system partitioning, various algorithms in use in this domain and criteria which guide system partitioning. Section 3 describes the problem domain, the testability model and the test strategies required for evaluating the system test cost. Section 4 applies this evaluation for hardware/software partitioning. A real piece of embedded system illustrates our approach. Section 5 proposes a generalization of the test cost measurement based on an axiomatic decomposition of a control flow structure. Conclusions and remarks are given in Section 6.

2. State of the art

Application-specific systems have important applications in communication, multimedia, consumer products, robotics and control systems. They often incorporate hardware and software together to meet performance and cost requirements. The design process is called "codesign" to distinguish it from the design of systems consisting exclusively of hardware. One of the major problems in codesign is finding the best hardware/software (HW/SW) compromize to realize the system: this is called HW/SW partitioning. To explain the problem, we must show its position with regard to the codesign flow and its interaction with the other steps. As illustrated in Figure 1, the first step in a codesign flow is system specification capture. This stage results in a functional specification of the system. Once the system specification is captured and validated, the next stage partitions the specification objects into hardware and software. In this case, estimators are used by

---

* AEROSPATIALE is the French partner of the AIRBUS consortium

the partitioning algorithm to choose the best partition from the huge number of possible ones. If a target architecture is a priori imposed by the strategy, this exerts considerable influence on the partitioning algorithm and solutions must respect this architecture (the dashed line in Figure 1 means the possible presence of the target architecture at this stage). In the refinement stage, interfaces between hardware and software parts are generated and added to the target architecture. Finally, software and hardware parts are synthesized and the target architecture is cosimulated (HW/SW simulation) to validate the proposed solution. Suggested changes are fed back to some or all of the previous stages.

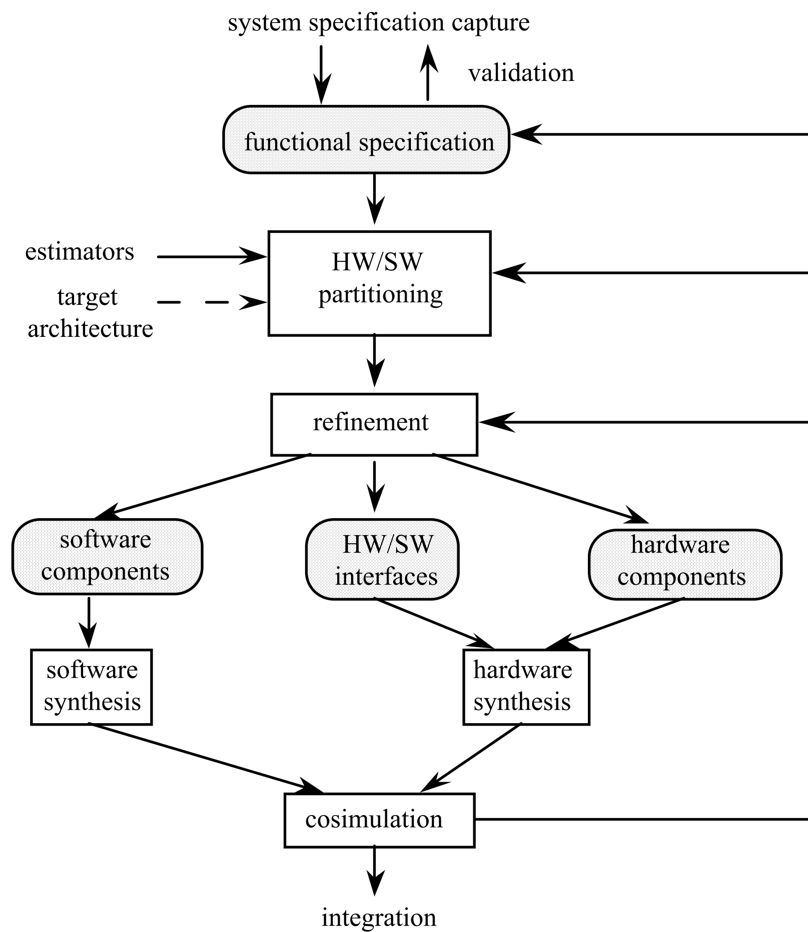

Figure 1: A codesign flow

Specification capture consists in defining desired system functionality which in turn is determined by the system's interactions with its environment. Afterward, the functionality is recursively decomposed into smaller components by creating a conceptual model of the system.

The result of this phase is a functional description which can be written in a formal language which allows the application of validation techniques such as simulation or verification. According to Kalavade and Lee [4] classification, two opposing approaches for system-level specification are emerging. One is the *unified* approach, which seeks a consistent semantics for specification of the complete system. In this case the description formalism must be powerful enough to describe system-level specification but not only software or hardware specifications. Accordingly, tools must support cosimulation to validate software and hardware parts. The second one is the *heterogeneous* approach, which allows the different parts of the system to be represented in different formalisms. For example, data-flow models to represent computation algorithms and FSM (Finite State Machine) models to represent controllers. However, the difficulty inherent in this approach resides in making the relation between the different formalisms and the link with the architecture. A common intermediate language might be a solution to this difficulty as proposed in ASAR project [12].

Once the functional specification of the system has been captured and validated, the problem of HW/SW partitioning is raised. It consists in choosing either a hardware or a software implementation for each of the different parts constituting the system. Generally, no fixed rules exist for performing this choice, but very often the hardware choice is made for performance reasons and software choice for cost and product-evolution reasons. According to Gajski and Vahid [13], in a HW/SW partitioning process we must consider the following issues: object granularity, design metrics, estimation and partitioning algorithms.

Object granularity defines the smallest indivisible functional objects used in the partitioning process. Granules could be jobs, processes, subroutines, loops, blocks of statements, statements, arithmetic operations or Boolean expressions. Design metrics are used by estimators to evaluate a partition quality which in turn is used by the partitioning algorithms to select a subset of possible partitions. The most often used metrics are performance and cost. Other metrics such as hardware size, communication rates, power consumption, package size, testability, reliability, design time, program size, data size, memory size could be used with the previous ones to improve partition quality or when some target architectures are a priori imposed. There are a variety of estimation tools and techniques suggested by research. All of them estimate speed and size either for hardware

[14-18] or software [19]. To obtain rapid hardware speed and size estimates, estimators may perform a rough synthesis of the hardware. For software, estimators can use a compiler into a processor instruction set to obtain execution-time and memory-size estimates.

Given a functional specification, estimators and perhaps a target architecture (depending on whether it is imposed before the beginning of partitioning process or not), the partitioning algorithm explores a subset of the possible partitions, that satisfies the imposed constraints. Different strategies are proposed in the literature to perform this task.

Gupta and De Micheli [1, 20] proposed a hardware-oriented strategy with a priori fixed monoprocessor target architecture. The functional specification is compiled into a hierarchical sequencing graph where vertices represent operations and edges represent inter-operation dependencies. This model is translated into concurrent tasks which constitute the input data for estimators and the partitioning algorithm. The initial solution is hardware-based excepting for ND (Nondeterministic Delay) operations. If it doesn't meet the required performances, no solution exists for this problem because performance with software is lower. Otherwise, starting with the initial partition, the algorithm iterates and explores new solutions by migrating hardware operations into software. A solution is conserved only if it is better than the previous one. This method has the advantage of always proposing a solution if it exists, however sometimes it gives a non-optimal solution as optimal because of local-minimum phenomenon.

Henkel and Ernst [2, 21] have developped a COSYMA tool for synthesizing codesign systems (a "cosynthesis" tool). HW/SW partitioning is performed by a software-oriented algorithm, i.e. starting with a software-based solution, the algorithm determines by simulation techniques the objects which violate the performance constraints to be implemented in hardware. Thus a maximum number of objects are implemented in software. The authors justify this with the many software properties that hardware lacks such as flexibility, debugging, high density of microprocessor memories, adapted compilers, etc. An a priori fixed monoprocessor target architecture is used to implement the system.

Kumar et. al. [22, 23] proposed a formal conception for codesign and a successive refinement partitioning algorithm. At the system-level, the functionality of the system is decomposed into several smaller functions and HW/SW refinement authorises the switch from system-level to

algorithmic-level. It consists in refining software and hardware parts separately from each other. For example, at the algorithmic-level, software components become sequences of instructions and hardware components fetch/execution units. At this level, a partition corresponds to an implementation choice. So, in this way the algorithm explores a large number of solutions for implementing a particular function. Using metrics and estimation values the algorithm determines the most appropriate solution to the problem.

Barros et. al. [24, 25] proposed a method to partition codesigned systems specified by the UNITY language. The basic idea of the partitioning algorithm is to classify functional objects with respect to their parallelism and independence (parallelism degree, mutual exclusion, data dependencies). The partitioning process begins by this classification and proceeds in a way that always try to increase the parallelism degree, reduce communications cost and minimize the area/speed function. The target architecture is generated and costs are computed and compared to previous solutions.

Gajski et. al. [13] proposed a partitioning strategy based on the SpecCharts formalism. In this case, system specification is modeled by a hierarchical and concurrent FSM. To each element of the SpecChart are associated its hardware and software implementations and the related characteristics. If not known, characteristics may be dynamically computed by estimators. The target architecture is a priori fixed and contains a single processor. The partitioning algorithm is based on a binary constraint-search (BCS) mechanism. It searches a solution that respects performance constraints and minimizes the hardware added to the processor.

The refinement stage consists in adding details about how interfacing hardware and software parts resulted from the partitioning stage. Many interfacing details are performed: memory access protocols are inserted to every part of the system description that accesses a variable in the memory, buses are generated to communicate data between system components, and finally arbiters are generated to resolve simultaneous access to shared resources such as buses and memories.

Once software, hardware and interfacing parts are defined, the synthesis process is applied. The software synthesis process consists first in converting complex system-level descriptions (concurrent tasks, "busy-waiting", scheduling) into traditional software programs which may be compiled by traditional compilers, and then in compiling the outcome of the first step into the

target processor instruction set or into a generic instruction set if the target processor is not yet known or its compiler is not available. The hardware synthesis process consists in applying high-level synthesis techniques to synthesize hardware parts into a target technology in order to be implemented on custom components such as ASICs or FPGAs.

Finally, cosimulation is performed on the target architecture to validate the proposed solution. In case the solution no longer satisfies the constraints because of interface insertion, suggested modifications are fed back into the previous steps as shown in Figure 1.

3. Test and testability of a data flow system

In this section, we focus on evaluating the test cost of a hierarchically specified codesign system. Test cost is defined as the number of test cases needed to test a system. Two parameters have a great impact on system test cost: test strategy and unit-level implementation choice. Each level of specification may use basic statements or operators as well as sub-specifications. The lower level specifications are called unit-level components which may be implemented in hardware or software. The proposed analysis allows the evaluation of system test cost based on the evaluation of unit-level component test costs.

3.1. The problem domain

The piece of software design presented in figure 2 serves as the main example throughout the paper (excepting for section 5 on related work). The whole software is embedded in an on-board control unit integrated in the Airbus A320 aircraft flight management system. Typical avionic systems have their major functions allocated to separate configurations of software and hardware. The software under consideration has been produced using a practical approach to handling a codesign system; the diagram shown below is the output from an intermediate level design phase. The design method is called CAS (Computer-Aided Specification). CAS is an in-house method of AEROSPATIALE's; it has evolved naturally from earlier design methods widely used in the avionics industry, in particular from the blueprinting methodology for designing and describing electronic devices. CAS has been used successfully for years on a wide range of avionic systems. Much as SADT [26], CAS consists of both techniques for performing system design and a process for applying these techniques. Each diagram is a network of interconnected boxes which can be

decomposed into lower-level diagrams. The result of a CAS activity is a sequence of hierarchically organized diagrams: the highest level diagram represents the whole system, whereas each lower-level diagram shows a limited amount of detail.

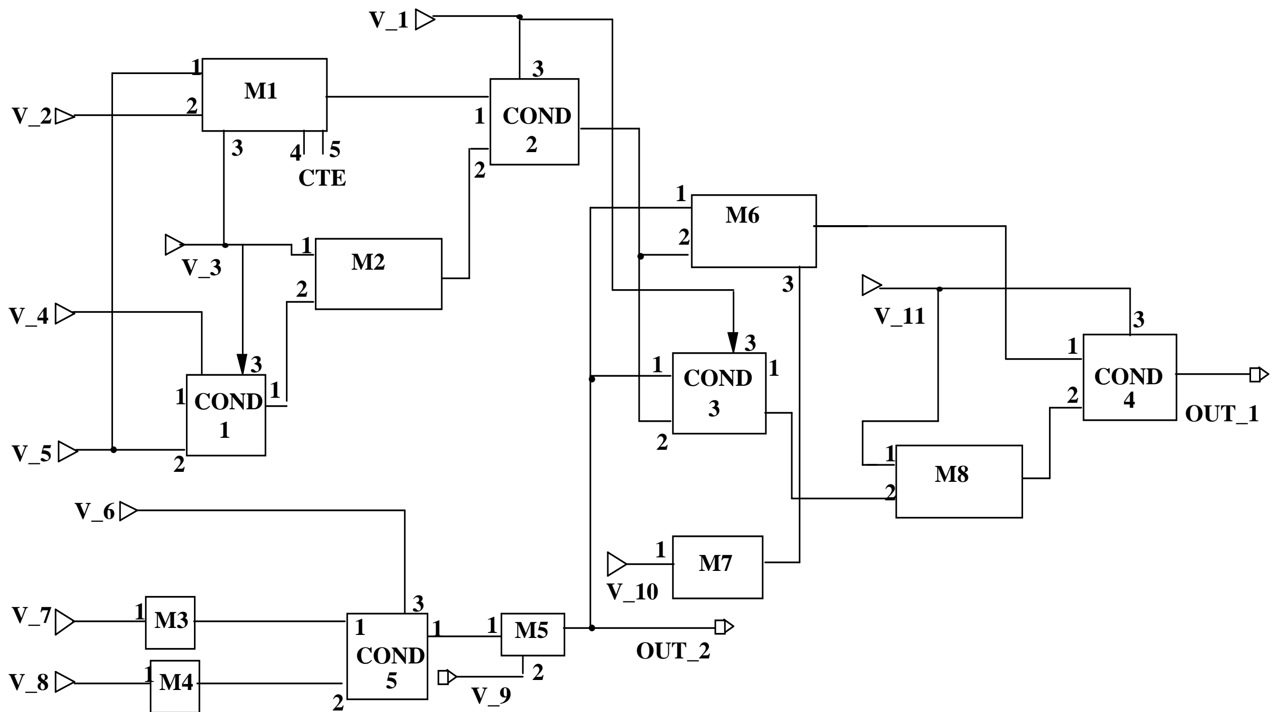

Figure 2: CAS diagram

As shown in Figure 2, the software contains two outputs OUT1 and OUT2 and uses elementary conditional operators: if-then-else, if-then and complex components (M1 to M8). Names and functions of the original specification have been changed.

3.2. *System modeling*

The principle of modeling information transfers consists in representing control and data flow aspects on the same graph. This is achieved by using a bipartite directed graph, called an *Information Transfer Graph*, in which two types of node occur: the modules which are associated with computations, and the transfers which characterize the mode of information transmission between these modules (the interconnection of modules models the ability to transmit information from one module to another one). For any computation node there can be only one information flow coming in or going out. So, an indegree (respectively outdegree) greater than one reflects a situation in which a selection among the entering (respectively exiting) flows has to be operated.

Thus, the computation nodes play the role of *or-nodes*, whereas the transfer nodes are similar to *and-nodes*. The information carried by the arcs arriving at a node is all that is necessary to activate the node. Similarly, every arc leaving a node contains all the information issued by the node.

Three basic modes of information transfer are necessary to encode CAS diagrams (see Figure 3) :

- junction mode: the data from source modules S_1, ..., S_n are all needed for the destination module D to be exercized;
- attribution mode: to be activated, the destination module D needs information from only one of source modules S_1, ..., S_n;
- selection mode: the information issued by source module S is sent to only one of destination modules D_1, ..., D_n.

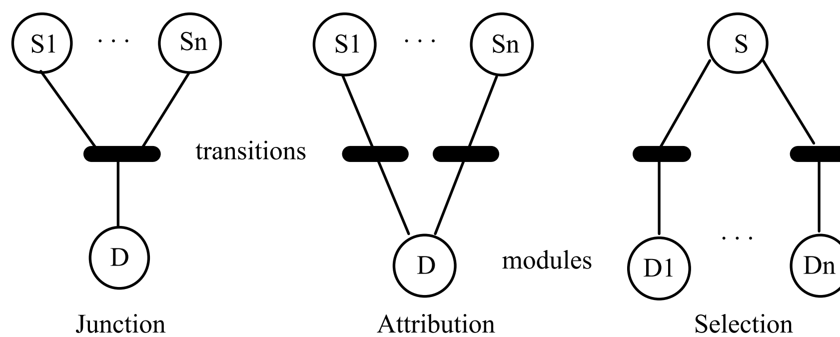

Figure 3. Information transfer modes

The construction process of the information transfer graph associated with a CAS diagram relies on a set of rules and schemas which are not detailed in this paper for the sake of brevity. The generation of the Information Transfer Graph is more precisely detailed in [27, 28].

The *Information Transfer Graph* is used to identify paths along which information is transferred. Therefore, this model contains all control points, like a conventional control graph, and also includes all relations between variables. This model is particularly suitable for data flow specification languages since it automatically includes the concept of paths between the definition and use of variables.

The information transfer graph groups the set of paths along which information is forwarded, from a subset of inputs towards a subset of outputs. These paths characterize the global specification functions. In the context of a functional specification language, a notion of flow in the

information transfer graph is proposed: a *flow* is an information path from inputs to one output through a set of modules. Thus, a flow characterizes a sub-function of the specification since it corresponds to one of the expressions of which the values define the output variable under consideration. Every flow is such that:

– whenever an output module of a junction is in the flow, then all its predecessors belong to the flow,

– whenever an output module of an attribution is in the flow, only one of its predecessors belongs to the flow,

– whenever an input module of a selection is in the flow, only one of its successors belongs to the flow.

A loop statement generates two flows, one exercizing the loop, and the other not. As a result, flows can be viewed as elementary functions which define the various output variables. This justifies the selection of test strategies which focus on the coverage of the modules included in these flows.

The Information Transfer Graph associated to the CAS diagram of Figure 2 is illustrated in Figure 4.

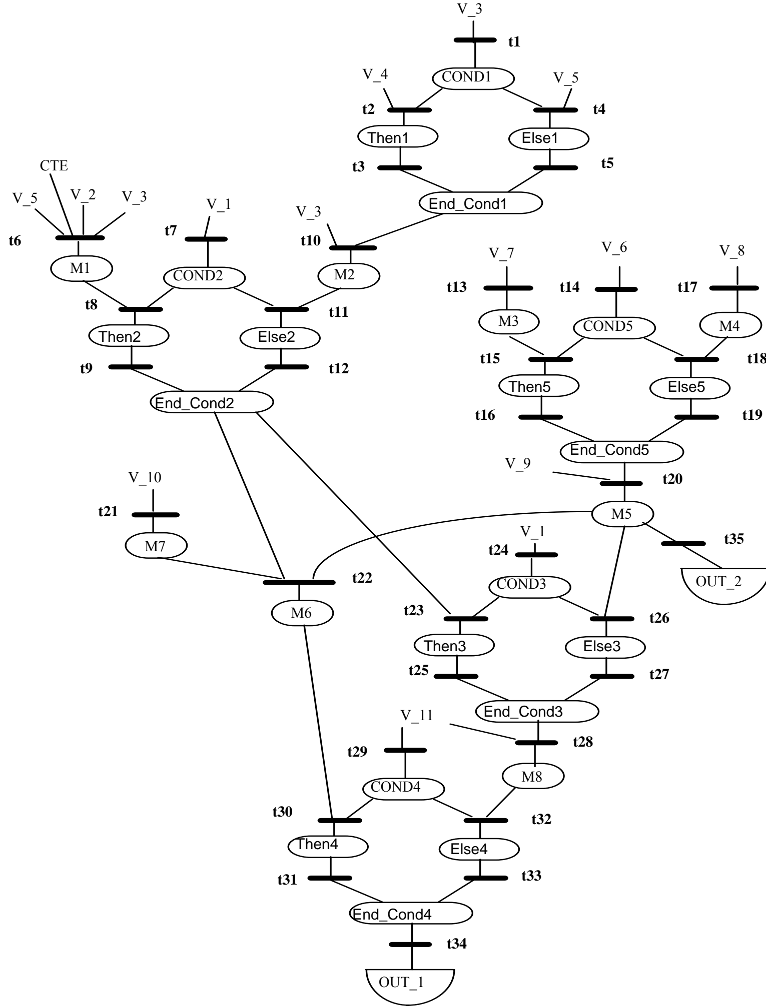

Figure 4. Modeling a CAS diagram into ITGraph

Note that modules stuck at "Then" and "Else" branches of an "If" statement are only mentioned to indicate the origin of information in the flow and have no consequence on the flow test cost as detailed in section 3.4.

In the example under consideration (see Figure 2), 13 flows are identified. For the sake of readability, we do not represent flows by a list of exercized transitions. They are listed below. Each flow F_i is noted as a set { list of all exercized modules | *output variable*}.

F1 = {M1, M3, M5, M5, M6, M7 | OUT1}

F2 = {Then1, M2, M3, M5, M5, M6, M7 | OUT1}

F3 = {Else1, M2, M3, M5, M5, M6, M7 | OUT1}

F4 = {M1, M4, M5, M6, M7 | OUT1}

F5 = {Then1, M2, M4, M5, M6, M7 | OUT1}

F6 = {Else1, M2, M4, M5, M6, M7 | OUT1}

F7 = {M1, M8 | OUT1}

F8 = {Then1, M2, M8 | OUT1}

F9 = {Else1, M2, M8 | OUT1}

F10 = {M3, M5, M8 | OUT1}

F11 = {M4, M5, M8 | OUT1}

F12 = {M3, M5 | OUT2}

F13 = {M4, M5 | OUT2}

3.3. Test strategies

Our objective is to propose structural system testing strategies which take advantage of the nature of the designs to provide some form of functional testing. The key task, then, is the identification of basic functional computational units. We argue that data flow analysis on data flow designs can be used to automatically recognize functions and ensure that they have been tested.

A test strategy is defined in this paper as a set of flows which have to be exercized in order to test a system with respect to a criterion and a development phase. A test strategy is a test specification. Such a test specification describes the function's inputs, the environment conditions that affect its behaviour, the outputs it produces, and the changes it makes to the environment. Since the test specifications are based entirely on the program's expected functional behaviour, they can be written as soon as the software specification is available.

In this study, we consider three testing strategies concerning the validation process: Start-Big, Start-Small and All-Paths strategies.

The All-paths strategy consists in an exhaustive test of all the possible paths through the graph associated with the CAS diagram. In contrast, the Start-Big strategy consists in choosing a minimum number of paths to guarantee the coverage of each edge of the graph. Consequently, the Start-Big corresponds to a test of the main functionality of the system and the All-Paths specifies exhaustive testing of the system. As an intermediary strategy between paths coverage and edges coverage, the Start-Small is a bootstrapping (incremental) method: the first test data set concerns a

minimal portion of software sufficient to run the subsequent tests. Each additional test set adds a small portion of software to the previously tested parts. When a fault is detected, the part in which it is located is one of those added for the next test set. The first error must be repaired before running the following tests. The Start-Small test strategy allows functions to be ordered and test paths to be determined among the set of flows. Start-Small strategy is an incremental test strategy. So, the Start-Small strategy is based on gradual coverage and correction of the modules by ordering flows from the smallest, covering the fewest modules, to the largest. The process is iterative and assumes that all modules in one flow have been corrected before activating the following flow. The process ends when all modules in the graph have been tested.

The All-Paths on the one hand, the Start-Small and Start-Big strategies on the other hand, are totally different since All-Paths guarantees the coverage of all the flows whereas Start-Small and Start-Big only guarantee the coverage of each module. These strategies are more precisely detailed in [29].

As an example, let us consider the sets of flows corresponding to each strategy based on the Figure 2 CAS diagram :

All-paths = {F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, F13}

Start-Small = {F1, F2, F3, F7, F10, F12, F13}

Start-Big = {F2, F3, F7, F10, F11}

The All-Paths strategy effectively exercizes all the possible flows and Start-Big the minimum subset to cover the model: Start-Small appears as an intermediary strategy.

3.4. Testability evaluation

In this section, we look at the testability attribute of a system. We define testability with respect to control-flow-based test data selection strategies and show how this may be measured as a property of the design.

Testability is concerned with three characteristics: generation of a test set, interpretation of the test results and fault repair (when faults are detected). The repair characteristic depends mainly on the diagnosis of detected faults since faults must be located before being corrected. We focus in this

paper on the two first points. The third, which concerns the diagnosability of the system has been studied in [9].

Therefore, we associate the test cost to the effort needed to generate a test strategy and interpret the results. In measuring the testing effort, we consider that:

- for a given flow, test generation and interpretation effort is a function of the modules exercized by the flow. The more “complex” the modules, the more the effort required to generate and interpret the corresponding test,
- the size of the test set increases with the number of flows needed for a given test strategy.

The ambiguous "complex" concept is related in this paper to the number of test cases needed to test the considered module (it corresponds to the *weight* of the module in the flow). Therefore, a flow generation and interpretation effort FGIE is considered as a function of flow module weights and can be interpreted by the following relation:

$$FGIE(i) = F(\{m_j\}_{j \in Modules}) \qquad (r1)$$

where m_j = weight of module j in flow i

The problem is to give a value to the module weights and to choose a suitable function for F. For a given flow, which consists of sequential nodes, the testing cost is considered to be equal to the testing cost of the most difficult node to be tested within the flow. So, the MAX function has been chosen for evaluating the flow generation and interpretation effort. These points are discussed in the next section.

Moreover, we consider that the number of test sets to be generated will increase with the number of flows. The global difficulty in implementing a test strategy can then be estimated as the sum of all FGIEs of flows that it induces on a given system specification. Thus the total Generation/Interpretation effort is defined as being equal to GIE:

$$GIE(S) = \sum_{i \in F_S} FGIE(i) \qquad (r2)$$

where F_S is the set of flows induced by the strategy S.

This technique permits an estimate of the test efforts for different test strategies: for example AllPaths, Start-Big or Start-Small as will be illustrated in the application example. However, the general idea of the approach remains to estimate the test cost for a specification level by using the test costs of the components of the lower level specifications.

4. System test cost

The principle of the approach consists in assigning two values for each unit level component significant to the number of test cases needed for testing its functionality: N_{hard} for the hardware test and N_{soft} for the software test. We evaluate then the system test cost by recursively computing the GIE for each intermediary specification level as presented in Figure 5. For each specification level, the test cost is computed with respect to a chosen test strategy. As explained in section 4.1, N_{hard} and N_{soft} for unit-level components are provided by the mutation-based testing method.

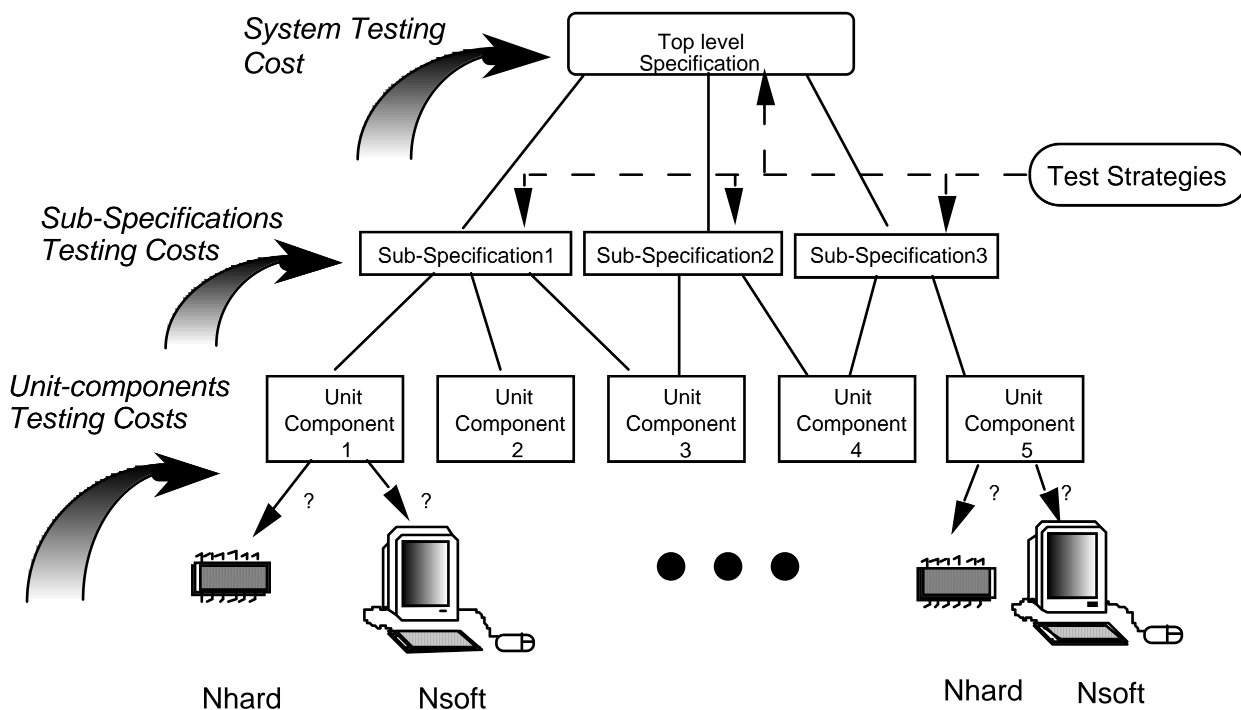

Figure 5. System Test Cost based on lower level Test Costs

4.1. Mutation analysis

Mutation analysis, originally proposed in 1978 [10], consists in creating a set of faulty versions or *mutants* of a program with the ultimate goal of designing a test set that distinguishes the program from all its mutants. Faults to be considered are those which occur most commonly in software programming. On the basis of the two hypotheses, the *coupling effect* and the *competent programmer*, a mutant is defined as a faulty version which differs from the original program by a simple, unique and syntactically correct change. This allows, in practice, faults to be modelled by a

set of *mutation operators* where each operator represents a class of software faults. To create a mutant, it is sufficient to apply its associated operator to the original program.

Figure 6 shows the VHDL functional description of the unit-level component “Component-5” with four overlapped mutants (this component is used twice as M6 and M8 modules in the main example). When displaying mutants, the convention is to show the original program with each mutated statement inserted immediately below the corresponding original statement. Each mutated statement represents a separate (mutant) program with the original statement replaced by the mutated statement. For example, the first mutant is generated by the ROR (Relational Operator Replacement) mutation operator which replaces the relational operator by another one, and the last mutant by the “No Operation” operator which replaces the statement by the “null” statement. A set of mutation operators related to VHDL is given in Annex 1.

```
   entity Component 5 is
     port(threshold:  in integer range -2**31 to 2**31-1;
          delta :     in -2**31 integer range to 2**31-1;
          data in :   in -2**31 integer range to 2**31-1;
          data out:   out -2**31 integer range to 2**31-1);
   end Component 5;

   architecture behavior of Component 5 is
   begin
     Component 5 P: process (threshold,delta,data in)
     begin
1      if data_in > threshold then
•      if data in <= threshold then
2        data_out <= data_in + delta;
•        data_out <= data_in - delta;
•        data_out <= - data_in;
•        null;
       elsif data in < threshold then
3        data out <= data in - delta;
       end if;
     end process Component 5 P;
   end behavior;
```

Figure 6: Functional VHDL description of Component-5.

The mutation analysis was basically proposed to evaluate the effectiveness or the relative adequacy of a test set with respect to a given fault model; a test set is relatively adequate if it distinguishes the original program from all its non-equivalent mutants. Otherwise, a *mutation score* is associated to the test set to measure its effectiveness in terms of percentage of the revealed non-equivalent mutants. It is to be noted that a mutant is considered equivalent to the original program

if there is no input data on which the mutant and the original program produce a different output. A benefit of the mutation score is that even if no error is found, it still measures how well the software has been tested giving the user information about the program in the absence of errors. It can be viewed in a way as a reliability assessment for the tested software. Moreover, it is worth noting that in recent works, mutation analysis is used itself to automatically generate the test data by means of a *constraint-based* technique.

In [11] the mutation analysis is detailed and an approach is proposed to adapt this technique for generating test data for VHDL behavioral descriptions. The basic idea behind the approach is to define a unified strategy that tests both the behavioral specification, seen as a software program, against (software) design faults as well as its hardware implementation against (hardware) manufacturing faults. For testing software faults, the mutation-based test set is already sufficient as the technique was originally proposed to test software programs. However, for sufficiently testing the hardware implementation, hardware characteristics must be taken into account. Therefore, the software test set needs to be enhanced by additional test vectors. At present, we consider bit-width characteristics of the hardware implementations to guide the enhancement process. As a result we obtain two test sets, T_{soft} and T_{hard}. T_{soft}, obtained before the enhancement process, is sufficient for testing the behavioral specification and T_{hard}, obtained after the enhancement process, is sufficient for testing the hardware implementation (Figure 7). N_{soft} and N_{hard} are the corresponding test set sizes, which are used as basic test costs in the system test cost evaluation.

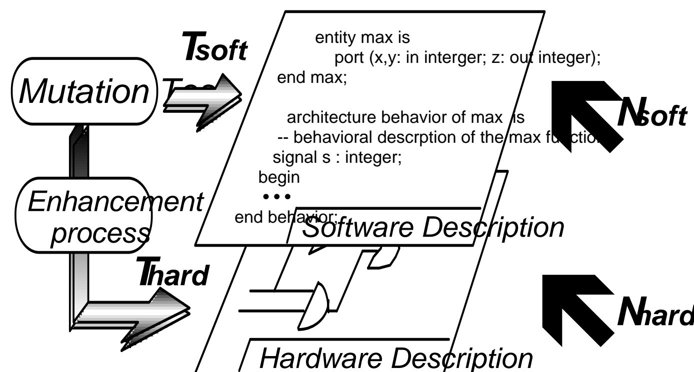

Figure 7. From mutation test to unit-components test costs (N_{hard} & N_{soft})

4.2. *System partitions test cost*

Once N_{hard} & N_{soft} are computed for each unit-level component and a test strategy is chosen for each specification, the system test cost is recursively computed from the lower level specification to the top level as follows:

- if the specification is a unit-level component, the reported value is N_{soft} or N_{hard} depending on the implementation choice.
- otherwise, relations r1 and r2 of section 3.4 are used to evaluate the test cost of the specification according to the selected test strategy. Each module of the specification corresponds to a subspecification or to a basic processing (like arithmetic, Boolean operators or units components). Weights of basic processing modules are estimated by mutation testing (a library is constituted). Weights of subspecifications modules are the reported test costs values from the lower level.

It is to be noted that the system test cost is equal to the GIE of the top level specification for a given partition. So, this approach results in a set of all possible partitions test cost.

Integrating testability aspects during the partitioning phase may have very high returns. Since the test cost provides information on the difficulty of testing the system with respect to a quality test criterion (coverage of modules, edges or paths for example), it has an impact on the real validation cost as well as the reliability of the system. Indeed, a system that is more testable than another (if comparable) is expected to easily reach a better fault-detection rate and consequently a higher reliability for the same validation effort.

To complete the partitioning process, the test cost criterion has to be mixed with other partitioning criteria, such as performances and cost. Each system is more or less safety-critical and depending on its real environment constraints the designer has to estimate the importance of each partitioning criterion. So, flexibility of a partitioning criterion is a highly desired feature. In fact, it has to be used in accordance with the designer appreciation and in parallel with other criteria. Our approach allows the designer to fix a priori some module's implementation (with respect to his own experience or to other specific criteria) and evaluate for the remaining components which of them can be hardware implemented with respect to a certain test cost constraint.

4.3. Application example

In this section, we apply the aforementioned approach to the Computer Assisted Diagram described in figures 2 and 3. It consists of a two level hierarchical specification. The lower level contains eight unit-level modules labelled from M1 to M8. However, only five modules perform different tasks. That is to say, there are five different unit-level components which may be used in various places in the specification. In this example, modules which instance a one unit-level component are: {M2, M4}, {M5, M7}, and {M6, M8}. Therefore, each subset will be assigned to the same hardware or software value that the original component will have throughout the partitioning process. The N_{soft} and N_{hard} values for each unit level component and the mapping to different modules are given in Table 1.

Unit-level Components	Comp1	Comp2	Comp3	Comp4	Comp5
Instances	{M1}	{M2, M4}	{M3}	{M5, M7}	{M6, M8}
N_{soft}	10	25	9	3	21
N_{hard}	25	85	17	9	80

Table 1 : Hardware and software test costs for unit-level components

The test costs for the All-Paths, Start-Big, and Start-Small strategies are given in Tables 2 respectively. They are presented as a function of the implementation choices for the unit-level modules (H for hardware, S for software, X when the implementation does not affect the result).

Comp1	Comp2	Comp3	Comp4	Comp5	GIE(All-Paths)	GIE(Start-Small)	GIE(Start-Big)
S	S	S	X	S	293	147	117
H	S	S	X	S	305	155	121
S	S	H	X	S	301	155	117
H	S	H	X	S	313	163	121
S	H	S	X	S	773	327	297
H	H	S	X	S	785	335	301
S	H	H	X	S	781	335	297
H	H	H	X	S	793	343	301
X	S	S	X	H	914	434	400
X	S	H	X	H	922	442	400
X	H	S	X	H	1009	504	415
X	H	H	X	H	1017	512	415

Table 2 : Test cost trends for All-Paths, Start-Small and Start-Big strategies

On the one hand, the relative efforts needed for applying each strategy as a function of the implementation choices can be compared thanks to Tables 2. As it was expected, the Start-Big test is the easiest to apply and the All-Paths test the most difficult. The Start-Small test implies an effort comparable to Start-Big even if it is more dependent on the little module implementation choices. Such a strategy seems to be a good compromize between the minimum test (Start-Big) and the exhaustive test (All-Paths).

On the other hand, analysis of Tables 2 reveals the unit components which have the main influence on the global test cost regardless of the chosen strategies. In grey, the three main categories of important implementation choices are outlined. The global system test cost depends on component 5 in the first degree (dark grey category) and component 2 in the second degree (light grey category). On the contrary, the component 4 implementation choice never influences the test cost. The other components have an intermediate and variable influence on the resulting system test cost with respect to the chosen strategy. For example, the component 3 implementation choice does not influence the test cost of the Start-Big strategy. As a result, the unit components may be ordered as a function of their respective influence on the global test cost: component-4 $\leq$

component-3 ≤ component-1 < component-2 < component-5. The relative importance on the test costs of these components could not easily be predicted. For example, component-2 has N_{soft} and N_{hard} greatest values while component-5 has the greatest impact on system testing cost. To conclude, such an analysis enables us to estimate the relative test costs for applying various strategies as a function of implementations choices. It could be useful to help testers choose a test strategy as a function of a given accepted test cost. Moreover, it highlights the components for which implementation is test-cost critical.

5. Related work: generalization to control flow systems

In order to obtain a more general definition of the testing effort, we propose an algorithm using the control flow structure of the system. It allows covering any kind of system, especially those designed using control flow description languages. Based on the axiomatic measurement, a rigorous evaluation of the test cost for each specification level is given which recursively account for the Test Costs of the subspecifications.

5.1. Axiomatic definition

To give an axiomatic definition of the test cost, we first need to detail the principles of the axiomatic decomposition of structured programs which were first introduced first by Ledgard and Marcotty in [30].

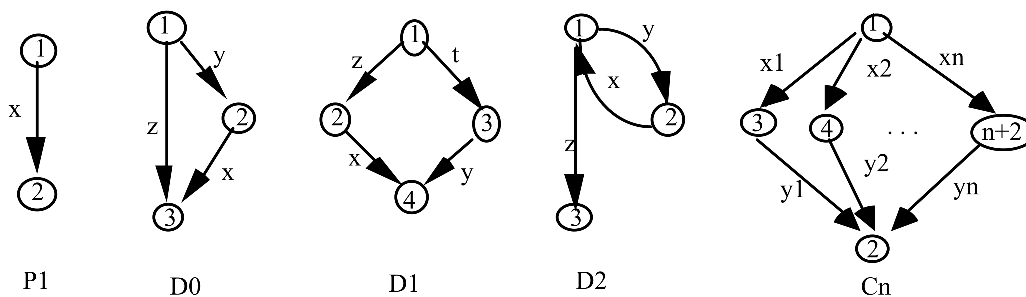

Figure 8: Prime flowgraphs

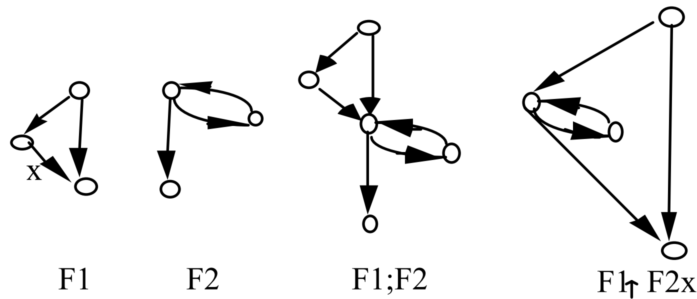

Figure 9: Decomposition operators

Flowgraph: A flowgraph G is a directed graph in which two nodes have special properties: the start node has zero indegree and has directed paths to all other nodes of G and the stop node has zero outdegree and every node has a directed path to the stop node. Two types of nodes are encountered in a flowgraph, the process nodes which have an outdegree of one and the predicate nodes which have an outdegree of two or more.

Path: A path is a sequence of successive edges, starting from the start node to the stop node.

The way in which flowgraphs can be defined axiomatically has been described by Prather [31], Fenton and Whitty [32]. Prather has proved that a flowgraph admits a *unique decomposition* into a sequence of sequentially irreducible flowgraphs and that any sequentially irreducible flowgraph admits a unique decomposition into a prime flowgraph nested with a set of subflowgraphs.

Two operators are used to perform this recursive decomposition: the sequencing operator (;) and the nesting operator ($\overline{\ }$):

Given two flowgraphs F1 and F2, a new flowgraph (F1; F2) can be produced by identifying the stop node of F1 with the start node of F2. The nesting operation (F1 $\overline{\ }$F2$_{\varepsilon}$) proceeds by replacing a process node of F1 by the start node of F2, deleting the unique arc ε outgoing from a process node of F1 and replacing the node it led to by the stop node of F2.

The recursive decomposition of the flowgraphs by sequencing and nesting operators leads to the decomposition tree. A prime flowgraph is a flowgraph which cannot be trivially decomposed by sequencing or nesting. The most common prime flowgraphs which can be encountered in a specification description are presented in Figure 8. For example, D0 corresponds to the if-then

statement, D1 to the if-then-else statement and Cn to a general multiple-condition statement (a case). Loops similar to the while statement are represented by the D2 prime flowgraph. For brevity's sake, we restrict ourselves to these main prime flowgraphs, since it would be easy to complete the axiomatic measurement of testability with extra prime flowgraphs. Figure 9 illustrates the operations of nesting and sequencing flowgraphs. Flowgraph F1 corresponds to a structure D0 and the x edge is the unique edge in which another flowgraph may be nested. F2 corresponds to structure D2.

Each specification level may be decomposed in this way and represented in a unique flowgraph. Figure 10 illustrates the modeling of the VHDL description of the *Overcome_Obstacle* routine specification which is a high level routine of the application example (see Annex2 for VHDL description) The expression of this flowgraph with sequencing and nesting operators is: $(D2 \; \overline{\ }(P1; P1)_x) ; (D_0 \; \overline{\ }(P1;P1)_x)$.

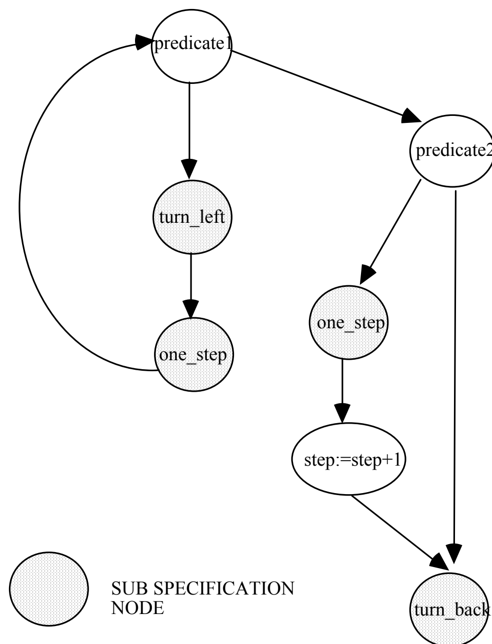

Figure 10: Overcome_obstacle specification

Defining axiomatically a measurement consists in assigning to each prime flowgraph a value and then combining these values through the nesting and sequencing operations. In fact, the following three rules are sufficient to define axiomatically a measurement:

R1 gives a value for the prime flowgraphs;

R2 explains how values are obtained when several flowgraphs are joined in sequence: written as f((F1; F2; ...; Fn)), where f is the rule R2 computing function.

R3 explains how values are obtained when several flowgraphs are nested into a prime flowgraph: written as f((P ¯ $F1_{x1}$, $F2_{x2}$, ..., Fn_{xn})), where f is the rule R3 computing function.

5.2. Axiomatic measurement of system test cost

Axiomatic measurement of the test cost is defined considering the control flow paradigm. The method for recursively computing the system test cost is the same as detailed in section 3.4.

Given a control flowgraph of a specification, the number of test vectors needed for testing the considered specification is computed as a function of the test cost value for each node. A flowgraph node corresponds to a subspecification, a unit-component or a basic process (such as arithmetic, boolean operators). The nodes values are defined as follows:

- subspecification nodes, the value is obtained by recursively applying the axiomatic algorithm to the subspecification,
- unit-level components, the value is either N_{hard} or N_{soft} depending on the implementation choice,
- basic arithmetic operator, five test vectors are generated, hence the value of the corresponding node is equal to 5. We have chosen this value because the mutation-based test generates, in general, five vectors for each arithmetic operator,
- predicate nodes, the value is 2 depending on whether the predicate is affected to the true value or not. Two values are available for the unit level which are significant of the number of test sets needed for testing the functionality: one (N_{hard}) for the hardware test and one (N_{soft}) for the software test.

Then, the number of test vectors needed for testing a specification is equal to the sum of the testing costs for each path within the flowgraph. The test cost of a path is obtained using the principles as for evaluating the test cost of a flow: it corresponds to the most difficult node to test

within the path. It is to be noted that, in case of loops, two paths are considered. One corresponds to a true value for the loop predicate and the other to a false one.

In the following, a formal definition of the algorithm is given.

Test Cost Algorithm

The test cost algorithm proceeds in two steps. First, the "Set of Paths Cost" SPC function builds up a set of path costs. It returns a set of integer values corresponding to the testing cost of each path of the specification. Then, the "Test Cost" TC function adds up all these costs in order to return the final Test Cost of the whole specification. We use the union operator ≈ to manipulate sets. The more particular operators H and □ are described below.

Let $\{V_1, V_2, V_3 ... V_n\}$ be a set of n integer values, the operator H has the following semantic:

$$H(\{V_1, V_2, V_3 ... V_n\}) = \sum_{1 \leq i \leq n} V_i$$

Let $\{V_{1.1}, V_{1.2}, V_{1.3} ... V_{1.n}\}$ be a set of n integer values and $\{V_{2.1}, V_{2.2}, V_{2.3} ... V_{2.q}\}$ another set of q integer values. The operator □ has the following semantic:

$$\{V_{1.1}, V_{1.2}, V_{1.3} ... V_{1.n}\} \square \{V_{2.1}, V_{2.2}, V_{2.3} ... V_{2.q}\} = \{max(V_{1.i}, V_{2.j}) \: / \: i \square [1..n] \text{ and } j \square [1..q]\}$$

In the following, Vi represents the value that a node in the flowgraph may have during the execution of the algorithm. For a unit-level component, this value may be attributed N_{hard} or N_{soft} according to the choice made by the user.

R1: Values of the prime flowgraphs:

$SPC(P1) = \{max(V1, V2)\}$

$SPC(D0) = \{max(V1, V2, V3)\} \approx \{max(V1, V2)\}$

$SPC(D1) = \{max(V1,V2,V4)\} \approx \{max(V1,V3,V4)\}$

$SPC(D2) = \{max(V1, V2, V3)\} \approx \{max(V1,V2)\}$

$$SPC(C_n) = \approx_{3 \leq i \leq n+2} \{max(V1, V2, V_i)\}$$

R2: Sequencing operator value

SPC(F1; F2) = SPC(F1) □SPC(F2)

R3: Nesting operator value

$SPC(P1 \overline{\ } F) = SPC(F)$

$SPC(D0 \overline{\ } F) = (\{max(V1, V3)\} \square SPC(V2)) \approx \{max(V1, V3)\}$

$SPC(D1 \overline{\ } F1_x, F2_y) = (\{max(V1,V4)\} \square SPC(F1_x)) \approx (\{max(V1, V4)\} \square SPC(F2_y))$

$SPC(D2 \overline{\ } F) = (\{max(V1, V3)\} \square SPC(F)) \approx \{max(V1,V3)\}$

$SPC(Cn \overline{\ } F1_{x1}, F2_{x2}, ..., Fn_{xn}) = \underset{1 \leq i \leq n}{\approx} (\{max(V1, V2)\} \square SPC(Fi_{xi}))$

The Test Cost of the whole specification is given by the relation: TC(F) = H (SPC(F))

5.3. Application example

In the case study we consider a robot for collecting precious objects in deep waters. The robot is provided with the necessary equipments such as a frontal sensor to detect obstacles and objects, a number of boxes to place fragile objects, a basket for non-fragile objects, and of course a hand to pick up objects and an electric battery to provide energy. During its motion, the robot signals each object detection and memorizes its position. This enables it to keep track of the object in case it cannot be picked up because the boxes or the basket are full or haven't enough space to contain the object. The motion algorithm consists in three steps which are repeated until the mission is finished or the battery discharged: advancing (*advance* routine) while no obstacle is encountered (*find_obstacle* routine). If an obstacle is encountered, it has to overcome it by turning around (*overcome_obstacle* routine). Finally, when the obstacle has been negotiated, it advances in a new direction. At any moment, the robot signals if an object is detected. In that case, it analyzes it (*object_analysis* routine) and when possible places it into box (if it is fragile) or into the basket. The turn_right, turn_left, turn_back are basic routines which change the current direction of the robot.

The unit-level routines are : *turn_right*, *turn_left*, *turn_back*, *advance* and *object_analysis*. Their corresponding N_{hard} and N_{soft} values are given in Table 3 .

	turn_right	turn_left	turn_back	advance	object_analysis
N_{soft}	3	3	3	17	11
N_{hard}	4	4	4	80	75

Table 3: Hardware and software testing costs for unit-level components

HW/SW implementation					
turn_right	turn_left	turn_back	advance	object_analysis	System Test Cost
S	X	S	S	S	84
S	X	H	S	S	85
H	X	S	S	S	87
H	X	H	S	S	88
S	X	X	H	S	163
H	X	X	H	S	164
S	X	S	X	H	498
S	X	H	X	H	499
H	X	S	X	H	501
H	X	H	X	H	502

Table 4: Case study testing costs by implementation choices

The results of applying of the algorithm to this case study are given in Table 4. In this example, there are three categories for the testing costs values (84-88, 163-164, 498-502). The lower one corresponds to software implementations for both advance and object_analysis routines. The 163-164 category corresponds to a hardware implementation of the advance routine and to a software implementation of the object_analysis routine. The higher one corresponds to the hardware implementation of the object_analysis routine. It is interesting to note that the routine with the greatest impact on the testing cost is the object_analysis routine. However, the advance routine, which has exactly the same order of testing cost values (and of hardware Area), is less important for the final testing cost of the system. Consequently, if the object_analysis routine is hardware implemented, the advance routine can be hardware implemented without affecting the testing cost of the system. In the same way, we note that the implementation choices for small routines

(turn_right, turn_left and turn_back) have little influence on the final result. The turn_left implementation choice does not affect it at all. Therefore, the unit components may be partially ordered as a function of their respective influence on the global testing cost: turn_left, (turn_right, turn_back), advance and object_analysis. The relative importance on the testing costs of these components could not easily be predicted. For example, object_analysis is less expensive to test than advance, and yet it has a greater impact on testing cost. In conclusion, we can say that the implementation choices can be testability-oriented.

6. Conclusion

The most significant point in this contribution is to integrate the test cost criterion in the process of system hardware/software partitioning. It provides to designer a global view on the system test cost as a function of the implementation choices of the unit-level components. Of course, the evaluation can be refined by taking into account the complexity of hardware/software interfaces (synchronization problems, data transfers, etc.). However, the actual evaluation is sensitive enough to catch the impact of each unit-level component on the system test cost. To our knowledge, such an approach, which guides the designer in a test-oriented partitioning of the system, is original.

References

[1] R. Gupta and G. De-Micheli, "System-level synthesis using re-programmable components," presented at EuroDAC, Brussels, Belgium, 1992.

[2] R. Ernst and J. Hankel, "Hardware-Software Codesign of embedded Controllers Based on Hardware Extraction," presented at International Workshop on Hardware-Software Co-Design, Estes Park, Colorado, 1992.

[3] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware/software codesign," *IEEE Design & Test of Computers*, vol. 10, pp. 6-15, 1993.

[4] A. Kalavade and E. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design & Test of Computers*, vol. 10, pp. 16-28, 1993.

[5] L. H. Goldstein and E. L. Thigpen, "SCOAP: Sandia Controllability / Observability Analysis Program," presented at IEEE Design Automation Conference, Minneapolis, 1980.

[6] C. Robach and S. Guibert, "Testability measures : a review," *International Journal of Computer Systems Science and Engineering*, vol. 3, pp. 117-126, 1988.

[7] R. S. Freedman, "Testability of Software Components," *IEEE Transactions on Software Engineering*, vol. 17, pp. 553-564, 1991.

[8] J. M. Voas and K. W. Miller, "Semantic Metrics for Software Testability," *J. Systems Software*, vol. 20, pp. 207-216, 1993.

[9] Y. Le Traon and C. Robach, "Towards a Unified Approach to the Testability of Co-designed Systems," presented at IEEE International Symposium on Software Reliability Engineering (ISSRE'95), Toulouse (France), 1995.

[10] R. DeMillo, R. Lipton, and F. Sayward, “Hints on Test Data Selection : Help For The Practicing Programmer,” *IEEE Computer*, vol. 11, pp. 34-41, 1978.

[11] G. Al Hayek and C. Robach, “From specification validation to hardware testing: a unified method,” presented at International Test Conference (ITC’96), Washington D.C. (USA), 1996.

[12] “The ASAR project: towards a multi-formalism framework for architectural synthesis,” presented at International Workshop on Hardware-Software Codesign, 1994.

[13] D. D. Gajski and F. Vahid, “Specification and Design of Embedded Hardware-Software Systems,” *IEEE Design & Test of Computers*, vol. 12, pp. 53-67, 1995.

[14] R. Jain, “MOSP: module selection for pipelined designs with multi-cycle operations,” presented at ICCAD, 1990.

[15] N. Dutt and J. Kipps, “Bridging high-level synthesis to RTL technology libraries,” presented at 28th DAC, 1991.

[16] F. Kurdahi and C. Ramachandran, “LAST: a Layout Area and Shape function esTimator for high level applications,” presented at EDAC, 1991.

[17] C. Ramachandran and F. Kurdahi, “TELE: a Timing Evaluator using Layout estimation for high level applications,” presented at EDAC, 1992.

[18] S. Narayan and D. Gajski, “Area and performance estimation from system level specification,” , technical report ICS-92-16, december 1992.

[19] J. Gong, D. Gajski, and S. Narayan, “Software estimation from executable specifications,” , technical report ICS-93-5, march 1993.

[20] R. K. Gupta and G. DeMicheli, “Hardware-Software Cosynthesis for Digital Systems,” *IEEE Design & Test of Computers*, vol. 10, pp. 29-41, 1993.

[21] J. Henkel, T. Benner, and R. Ernst, “Hardware-Software cosynthesis for microcontrollers,” *IEEE Design & Test of Computers*, vol. 10, pp. 64-74, 1993.

[22] S. Kumar, J. Aylor, B. Johnson, and W. Wulf, “A framework for hardware/software codesign,” presented at International Workshop on Hardware-Software Codesign, 1992.

[23] S. Kumar, J. Aylor, B. Johnson, and W. Wulf, “Exploring hardware/software abstractions and alternatives for codesign,” presented at International Workshop on Hardware-Software Codesign, 1993.

[24] E. Barros, W. Rosensteil, and X. Xiong, “Hardware/Software partitioning with UNITY,” presented at International Workshop on Hardware-Software Codesign, 1992.

[25] E. Barros, W. Rosenstiel, and X. Xiong, “A Method for Partitioning UNITY Language in Hardware and Software,” presented at EURO-DAC, Grenoble, France, 1994.

[26] D. T. Ross and K. E. Schoman, “Structured analysis for requirements definition,” *IEEE Transactions on Software Engineering*, vol. 3, pp. 6-15, 1977.

[27] C. Robach, P. Malecha, and C. Michel, “CATA: A Computer-Aided Test Analysis System,” *IEEE Design & Test of Computers*, vol. 1, pp. 68-79, 1984.

[28] C. Robach and P. Wodey, “Linking design and test tools: an implementation,” *IEEE Transactions on Industrial Electronics*, vol. 36, pp. 286-295, 1989.

[29] Y. Le Traon and C. Robach, “From Hardware to Software Testability,” presented at IEEE International Test Conference (ITC’95), Washington D.C, 1995.

[30] H. F. Ledgard and M. Marcotty, “A Genealogy of Control Structures,” in *Communications of the ACM*, vol. 18, 1975, pp. 629-639.

[31] R. E. Prather, “Hierarchical Metrics and the Prime Generation Problem,” *Software Engineering Journal*, vol. 8, 1993.

[32] N. E. Fenton and R. W. Whitty, "Axiomatic approach to Software Metrication through Program Decomposition," *The Computer Journal*, vol. 29, pp. 330-339, 1986.

Annexes

Annex 1: Mutation operators

Type	Description
AOR	Arithmetic Operator Replacement
ABS	ABSolute value insertion
CR	Constant Replacement
CVR	Constant for Variable Replacement
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement
NOR	No Operation Replacement
VCR	Variable for Constant Replacement
VR	Variable Replacement
UOI	Unary Operator Insertion

Mutation operators set for VHDL functional descriptions

Description of the functionality of each of the mutation operators:

AOR: Replaces occurrences of "+" by "-" and vice-versa.

ABS: Precedes arithmetic expressions and subexpressions with unary operators ABS and NEGABS. ABS computes the absolute value of the expression; NEGABS computes the negative of the absolute value.

CR: Constant values are slightly modified to emulate domain perturbation testing. The change to the value depends on the constant type. Each constant of type *integer* is both incremented by one and decremented by one. Each constant of type *bit-vector* is replaced by its 1's complement. Each *boolean* is replaced by its complement.

CVR: Replaces a variable by every compatible constant in the VHDL description.

LOR: Each occurrence of one of the logical operators (*and*, *or*, *nand*, *nor*, *xor*) is replaced by each of the other operators; in addition, the expression is replaced by TRUE and FALSE.

ROR: Each occurrence of one of the relational operators (<, >, <=, >=, =, /=) is replaced by each one of the other operators. In addition, the expression is replaced by TRUE and FALSE.

NOR: Replaces each statement by the *Null* statement.

VCR: Replaces a constant by every compatible variable in the program.

VR: Replaces a variable in the VHDL program by every compatible variable in the program.

UOI: Each arithmetic expression is negated, incremented by one and decremented by one. Each logical expression is complemented.

Annex 2: Robot VHDL description

```
-----------------------------------------------------------
--
--       ROBOT "description"
--      _____________________
--
--
--     The following VHDL program consists of a card description which controls
-- all the activities performed by a robot destined for selective-object
-- gathering. For example, this robot can search and gather precious objects in
-- deep waters. It is essentially equipped by a frontal sensor by which it
-- detects obstacles and objects, by a number of boxes to hold fragile objects
-- in and a basket for non-fragile objects. It is of course equipped by a hand
-- to pick objects up and a battery to provide him by energy.
-- The robot signals each object detection and memorizes simultaneously
-- its positions in order to keep trace to the object in case it
-- cannot pick it up because the boxes and the basket are full or haven't
-- enough space to put the object in.
--
-----------------------------------------------------------

library IEEE;
use     IEEE.std_logic_1164.all;
use     IEEE.std_logic_signed.all;

package all_functions is

--CONSTANTS
constant End_of_mission:                          integer:=0;
constant Boxes_are_full:                          integer:=1;
constant Fragile_object_picked_up:                integer:=2;
constant Fragile_object_abandoned:                integer:=4;
constant Basket_is_full:                          integer:=5;
constant Object_picked_up:                        integer:=6;
constant Object_abandoned:                        integer:=7;
constant OBSTACLE :                               integer:= 2;
constant OBJECT :                                 integer:= 1;
constant Max_steps :         integer:= 100000;

--TYPES
type position is array(1 to 2) of integer;
type direction is (NORTH, EST, SOUTH, WEST);
type object_nature is (FRAGILE, NON_FRAGILE);-- etc

--PROCEDURES & FUNCTIONS
procedure turn_right( d : inout direction);
procedure turn_left( d : inout direction);
procedure turn_back( d : inout direction);

procedure advance( d:in directin;
                   p:inout position);

procedure object_analysis( d:in directon;
          p:in position;
          o_nat:in object_nature;
          object_volum:in integer;
          box_size:in integer;
          empty_boxes:inout integer;
          empty_basket:inout integer;
          object_position :out position;
          mission_status:out integer);

procedure one_step(d:inout directtion;
                   p:inout position);

procedure Overcome_obstacle(d:inout direction;
                            p:inout position;
                            steps:inout integer);
```

```
procedure find_obstacle( d:inout direction;
                        p:inout position;
                        steps:inout integer);

end all_functions;

package body all_functions is

procedure turn_right( d : inout direction) is
begin
  d:=(d + 1) mod 4;
end;

procedure turn_left( d : inout direction) is
begin
  d:=(d - 1) mod 4;
end;

procedure turn_back( d : inout direction) is
begin
  d:=(d + 2) mod 4;
end;

procedure advance( d:in directin;
        p:inout position) is
begin
  case d is
    when NORTH   => p(1):=p(1) + 1;
    when EST  => p(2):=p(2) + 1;
    when SOUTH   => p(1):=p(1) - 1;
    when WEST => p(2):=p(2) - 1;
  end case;
end;

procedure object_analysis( d:in directon;
          p:in position;
          o_nat:in object_nature;
          object_volum:in integer;
          box_size:in integer;
          empty_boxes:inout integer;
          empty_basket:inout integer;
          object_position :out position;
          mission_status:out integer) is

begin
  if o_nat=fragile then
     if empty_boxes<=0  then
       if empty_basket<=0 then
       mission_status:= End_of_mission;
       else
       mission_status:= Boxes_are_full;
       end if;
     else
       if object_volum<=box_size then
       mission_status:= Fragile_object_picked_up;
       empty_boxes := empty_bxes - 1;
       else
       mission_status:= Fragile_object_abandoned;
       end if;
     end if;
     object_position:=p;--memorize_object_position
  else --o_nat is not gragile
     if empty_basket<=0 then
       if empty_boxes<=0 then
       mission_status:= End_of_mission;
       else
       mission_status:= Basket_is_full;
       end if;
     else
```

```
      if object_volum<=empty_basket then
      mission_status:= Object_picked_up;
      empty_basket := empty_basket - object_volum;
      else
      mission_status:= Object_abandned;
      end if;
    end if;
    object_position:=p;--memorize_object_position
  end if;
end;--procedure object_analysis

procedure one_step( d:inout directtion;
                p:inout position) is
begin
--sensor is a function which reads the sensor state and returns
--an integer indicating the nature of the object in the vision field
  while sensor()=OBSTACLE loop
    turn_right(d);
  end loop;
  if sensor()=OBJECT then
    object_analysis();
  end if;
end;

procedure Overcome_obstacle( d:inout direction;
                            p:inout position;
                         steps:inout integer) is
variable pi:position;
variable di:direction;
begin
pi:=p;--save the initial position
di:=d;--save the initial direction
while (p<>pi) and steps<max_steps loop
  turn_left(d);
  one_step(d,p);
  steps:=steps + 1;
end loop;
if steps<max_steps then
  one_step(d,p);
  steps:=steps + 1;
end if;
turn_back(di);
d:=di;
end;

procedure find_obstacle( d:inout direction;
                       p:inout position;
                       steps:inout integer) is
begin
  while sensor()<>OBSTACLE and steps < max_steps loop
    advance(d,p);
    steps:=steps + 1;
    if sensor()=OBJECT then
      object_analysis();
    end if;
  end loop;
  turn_right(d);
end;

end all_functions;

-------------------------------------------------------

library ieee;
use     ieee.std_logic_1164.all;
use     work.all_functions.all;

entity robot_ent is
  port (start : in std_logic;
```

```
       mission_status: out integer);
end robot_ent;


architecture robot_body of robot_ent is

signal d:direction;
signal p:position;
signal steps:integer;
signal m_status:integer;

begin

START_P: process (start)
        constant ORIGIN:position:=(0,0);
  begin
  if start=1 and start'event then
    d<=NORTH;
    p<=ORIGINE;
    steps<=0;
  end if;
  end process START_P;

MAIN_P: process (d,p,steps)
  begin
    if steps<max_steps and m_status!=End_of_mission then
      find_obstacle(d,p,steps,m_status);
      overcome_obstacle(d,p,steps,m_status);
    end if;
  end process MAIN_P;

mission_status<=m_status;

end robot_body;
```